WHERE SHOULD A SKILL EDIT LIVE? SCOPED ROOT EVOLUTION IN HIERARCHICAL AGENT SKILL LIBRARIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving agents turn execution experience into edits to a skill library, but most edit locally, so what they learn stays where it was written. In a hierarchical library only the shared *root skills* can carry an edit to tasks that did not produce it; for the same reason a root edit's scope is the whole subtree, so it can repair the tasks it was written for while silently breaking a sibling, which admission rules that score only the proposing tasks or the subtree mean cannot see. We recast root-level evolution as an *edit-placement* problem with a single rule: an edit belongs in the root only if no evaluated descendant regresses under paired execution, and otherwise in the leaves of the descendants it helps. SCOPEEVO instantiates the rule with root-aware revision, a subtree gate that promotes, localizes, or discards each candidate, and release recursion, and treats the leaf editor as a pluggable single-skill optimizer. On SkillsBench it further improves a library already evolved by a strong baseline, and on SearchQA and DocVQA it matches the strongest baseline with fewer editor calls and a more compact library. Its promoted roots repair tasks the loop never saw, and its refused root edits are recovered in the leaves rather than lost. We further show why a root edit needs subtree-wide validation while a localized edit does not, and that what a gate can guarantee from a few executions is bounded by the coverage of its validation set, which more runs cannot extend.

1 INTRODUCTION

Self-evolving agents turn execution experience into reusable procedural knowledge stored in a skill library, improving without parameter updates (Wang et al., 2023; Li, 2026). Most such systems evolve *locally*: a trajectory on one task yields an edit to the skill that task used. Local edits are cheap to validate, but what they learn stays where it was written (Xiong et al., 2026; Wang et al., 2026; Pu et al., 2026). In a hierarchical library, only the shared *root skills*, the top-level entries inherited by every skill beneath them, can carry an edit to tasks that did not produce it. Roots must evolve.

Precisely this makes evolving them hazardous: *the scope of a root edit is the whole subtree* (Figure 1(a)). A leaf edit changes one task and is validated on it alone; a root edit changes the descendants it was written for and every other descendant beneath the root, and nothing constrains the sign of the second part. Admission rules that score only the proposing tasks (LE-SRC) or the subtree mean (LE-AGG) are blind to it, so an edit that repairs one descendant and breaks a sibling passes both. This is not rare: on a QA subtree with two descendants, SearchQA and DocVQA, 11 of 16 candidate root edits move them in opposite directions on the validation set, and LE-SRC admits six of these, LE-AGG three (Figure 1(b)). The aggregate rule can raise the mean while lowering one descendant (Remark 1).

The failure is not that such edits are bad. An edit that helps the descendants it was written for while breaking a sibling is often correct knowledge lodged at the wrong level: a fact about some tasks, written where it applies to all. The question is therefore not *whether* to keep an edit but *where it should live*. Our answer is a rule: an edit belongs in the root only if no evaluated descendant of that root regresses under paired execution; otherwise it belongs in the leaves of the descendants it helps. Rejection becomes relocation.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

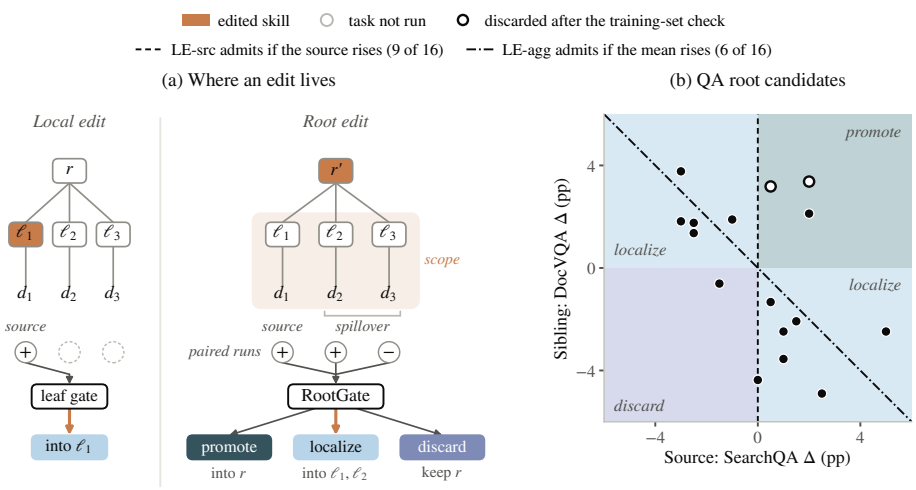

Figure 1: Where a root edit's effect lands. (a) A leaf edit changes one task; a root edit changes every descendant in its scope, and its spillover onto the siblings can have either sign. ROOTGATE runs paired executions on all of them and reads the placement from the same runs (here: localize). (b) The 16 candidate root edits of our QA subtree on the validation set, by their change on the source (SearchQA) and the sibling (DocVQA). Shading: ROOTGATE's exit; vertical and diagonal lines: admission boundaries of LE-SRC and LE-AGG.

Placement must be decided from execution: the object under test is an LLM policy, runs disagree, and the loop has a few executions. SCOPEEVO obtains that signal with three components. Root-aware revision conditions the editor on the current root, the evidence of every descendant in the subtree, earlier placement outcomes, and requires every proposed rule to state when it applies. ROOTGATE executes each descendant on paired inputs under the parent and the candidate library, since nothing outside the subtree can change, and resolves the candidate as promoted into the root, localized into the leaves of the descendants it helps, or discarded, reusing its own executions. Release recursion makes each accepted library, a promoted root or an accepted leaf, the parent of the next round. SCOPEEVO is a placement framework rather than a skill optimizer: it decides where an edit lives and leaves how the edit is written to a pluggable editor, so existing skill-optimization methods can serve as its leaf editor.

This paper makes four contributions. (1) *Problem.* We pose *edit placement* for hierarchical skill libraries and give a spillover decomposition that separates the descendants an edit was written for from the rest of the subtree and exposes collateral regressions under proposal-only and aggregate admission. (2) *Method.* We present SCOPEEVO, whose three-way placement is read from the gate's own executions and whose leaf editor is any single-skill optimizer. (3) *Theory.* We characterize placement under exact scores and faithful localization, derive miss rates for a fixed-budget, zero-margin gate, and establish the different validation scopes of root and leaf edits. (4) *Experiments.* On SkillsBench (Li et al., 2026b), SCOPEEVO improves a library that CoEvoSkills (Zhang et al., 2026b) has already evolved; on SearchQA and DocVQA it matches SkillOpt (Yang et al., 2026c), a strong single-skill optimizer, with fewer editor calls and less than half the skill text. Its promoted roots transfer to tasks the loop never saw, its refused root edits are recovered in the leaves rather than lost, and it combines the transfer of root placement with the task-specific gains of leaf placement, which neither scope achieves alone.

2 RELATED WORK

Skill libraries and their hierarchies. Agents that improve without weight updates externalize what they learn (Li, 2026), from Voyager's verified programs and Reflexion's verbal memory (Wang et al., 2023; Shinn et al., 2023) to workflows and skills induced from trajectories (Wang et al., 2024; Zhou et al., 2026; Qiu et al., 2026). A second generation validates before it writes, on a validation score (Yang et al., 2026c), a verifier (Alzubi et al., 2026; Zhang et al., 2026b), counterfactual

utility (Yang et al., 2026b), or a learned curator (Ouyang et al., 2026). A third line gives the library structure: abstraction levels (Xiong et al., 2026; Wang et al., 2026), retrieval hierarchies (Xia et al., 2026; Li et al., 2026a), typed graphs (Pu et al., 2026), a library shared across users (Ma et al., 2026), or SkillCommit's bottom-up expansion of instance-specific patches into broader skills, which replays a candidate on its source instances and can narrow it to those that support it (He & Yang, 2026). Their hierarchies are granularity levels or retrieval indices rather than inheritance, and none validates an entry on every task that will read it. We take the topology as given and ask, for an edit to an existing shared root, *at which level it should be written*, comparing every evaluated descendant, proposing or not, with the current parent, and rewriting a refused edit into private leaves.

Optimizing shared instructions. A root skill is a shared instruction, and a large literature optimizes such text by textual gradients, LLM-proposed rewrites, evolutionary search, compiled programs, or reflective evolution (Pryzant et al., 2023; Yang et al., 2024; Fernando et al., 2023; Guo et al., 2024; Khattab et al., 2023; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2024; Cheng et al., 2024; Agrawal et al., 2026), with analogues for workflows, system prompts shared across task types, playbooks, and cluster-level skills (Zhang et al., 2025; 2026d;e; Yang et al., 2026a). Across these methods the location of the text is fixed, so a candidate either replaces it for every consumer or is dropped, and acceptance is set-level: a mean over tasks or examples, a Pareto frontier that keeps candidates trading some examples for others, or a replay check, none of which requires that no consumer end up worse off. MAMuT comes closest by choosing between node-level and system-level optimization, but fixes that choice before optimization begins (Herrera et al., 2026). We differ not in how a candidate is written but in how it is accepted and where it lives: a root edit is promoted only if no evaluated descendant regresses and the subtree mean rises, and one that fails is moved into the leaves of the descendants it helped, a narrower scope that a flat prompt does not have.

Validation-gated updates and release control. Gating an update on re-execution rather than on the proposing trajectory's own judgment is now common, in release engineering (Zhang, 2026), memory writes (Xu et al., 2025; Chen et al., 2026), skill edits under a regression budget (Moll et al., 2026), self-modifying agents (Zelikman et al., 2023; Robeyns et al., 2025; Zhang et al., 2026c;a), and provisional rules that must not regress protected cases (Tayebati et al., 2026). Audits show why: evolution over a task stream can regress (Zhang et al., 2026f), and aggregate accuracy hides regressions in previously correct behaviour (Li & Zhu, 2026) that per-category suites catch, much as regression test selection does in software engineering (Chishti et al., 2026; Yoo & Harman, 2012). These gates are binary over one artifact; ROOTGATE reads an outcome per descendant of an inheritance hierarchy and has a third exit, and since its subject is a stochastic policy, it is a detector with computable miss rates and a coverage limit rather than a pass/fail oracle.

3 PROBLEM FORMULATION

Library and delivery. A hierarchical skill library L is a set of text nodes: root skills r and, beneath each, its descendants $\mathcal{D}(r)$, each a task d with an optional leaf skill ℓ_d. A fixed procedure ϕ assembles the execution context of task d from two nodes, $c_d(L) = \phi\big(r(d), \ell_d\big)$, where $r(d)$ is the root above d: a shared and a private skill package for the coding agent, a shared block and a task block of the system prompt for the QA executor. A frozen stochastic policy π executes d on an input x under $c_d(L)$, and a frozen scorer returns $s_d(L, x) \in [0, 1]$. Only node texts change between libraries, so a library that differs from another in one node can change score distributions only for tasks that read that node. We write $J_d(L; X)$ for the mean of s_d over inputs X and $J(L; X)$ for the equal-weight mean over descendants. Each descendant has a training set, which supplies evidence to editors, a validation set V_d, on which placement is decided, and a test set, scored once on final libraries; V_d is meant when X is omitted.

Root edits and spillover. An edit e to r yields the *candidate library* $L' = L[r \leftarrow r \oplus e]$, whose *parent* is L. A *paired execution* runs the same inputs under both and reports, per descendant, $\Delta_d(e) = J_d(L') - J_d(L)$ and the numbers of inputs that improve and regress, n_d^+ and n_d^-. With $\mathcal{P}(e) \subseteq \mathcal{D}(r)$ the *proposing set*, the descendants whose failures motivated e,

$$\Delta(e) \;=\; \frac{1}{|\mathcal{D}(r)|}\Big[\underbrace{\sum_{d \in \mathcal{P}(e)} \Delta_d(e)}_{\text{proposing descendants}} \;+\; \underbrace{\sum_{d \in \mathcal{D}(r) \setminus \mathcal{P}(e)} \Delta_d(e)}_{\text{other descendants}}\Big]. \tag{1}$$

Nothing constrains the sign of the second sum: a rule that repairs the proposing descendants may state a convention the others violate. Local validation executes only $\mathcal{P}(e)$ and never sees this sum; an aggregate score sees its total but not its distribution over descendants.

Edit placement. An edit can be written into r, which changes every descendant's context, or into the leaves of a subset $S \subseteq \mathcal{D}(r)$, as $\ell_d \oplus e_d$ with e_d the part of e relevant to d, which changes exactly S. Writing $\Omega(r) = \{r\} \cup 2^{\mathcal{D}(r)}$ for the set of placements, $\Delta_d(e;\omega)$ for the change on d under placement ω, and $\Delta(e;\omega)$ for its mean over the subtree, we seek

$$\max_{\omega \in \Omega(r)} \Delta(e;\omega) \quad \text{s.t.} \quad \Delta_d(e;\omega) \geq 0 \quad \forall\, d \in \mathcal{D}(r). \tag{2}$$

Root placement reaches descendants without a separate leaf write; leaf placement reaches only S and cannot violate the constraint elsewhere. The two admission rules in common use are relaxations of this problem: LE-SRC drops the constraint and writes e into the root iff $\sum_{d \in \mathcal{P}(e)} \Delta_d(e) > 0$, and LE-AGG replaces it by the aggregate and writes e into the root iff $\Delta(e) > 0$. Both are functions of the same paired executions as our rule, so they can be replayed at no execution cost. What makes the problem hard is that its constraint is not given but must be measured. $\Delta_d(e)$ outside $\mathcal{P}(e)$ exists only if d is executed under the edited root, and it is estimated from a stochastic policy with a few paired runs, so any placement rule is a detector with a miss rate rather than an oracle; a descendant with no validation input, or at ceiling under both libraries, returns no signal at any number of runs; and decisions compound, so the constraint must hold along the release chain, which an increase in the aggregate alone does not ensure.

4 METHOD: SCOPEEVO

SCOPEEVO runs *root events* and *leaf events* over a library L_k (Figure 2; pseudocode in Appendix A). A root event asks a root editor for a candidate written against the whole subtree, has ROOTGATE execute every descendant on its validation set under parent and candidate, and places the edit by a three-way rule that is the finite-sample form of the placement problem; an edit refused from the root is rewritten into the leaves it helped and validated there alone. Leaf events revise leaves with a pluggable leaf editor. Every accepted library, a promoted root or an accepted leaf, becomes the parent against which the next candidate is gated, so the no-regression constraint holds along the release chain and not only against the initial library: no event can trade a detected regression for a higher mean, which is what aggregate admission does at every round of a drift, and a promoted root can be edited again.

Root-aware revision. A gate can detect a shared cause but cannot create one, so the proposal side must see the whole subtree. The root editor receives the current root, every descendant's *contract* (its task statement and the output its scorer accepts) together with that descendant's leaf, bounded evidence from the training sets (failed and successful runs with their verifier messages or scored answers), and the outcome of the previous placement for this root: after a discard, the rejected candidate with its paired outcome; after a promotion, the runs of the promoted version; after a localization, the note that a rule has been localized to d and must state its applicability condition to stay in the root. The editor must identify the exact error in each failed run, separate rules that hold for every descendant from rules tied to one contract, and attach an applicability condition to every rule it writes; the last requirement keeps a root general rather than tailored to the tasks at hand, and the feedback of placement outcomes tells the editor which of its rules the subtree accepted. Edits are bounded in number and size (Appendix B), NOCHANGE is a legitimate answer, and the editor never sees validation or test results, nor the unseen tasks.

ROOTGATE. The candidate differs from its parent in one node, so score distributions can change only on $\mathcal{D}(r)$, the tasks that read the edited root. ROOTGATE executes exactly those descendants, each on V_d under both libraries, reads $\mathcal{B}(e) = \{d : \Delta_d(e) > 0\}$ and $\mathcal{R}(e) = \{d : \Delta_d(e) < 0\}$ over the evaluated descendants ($V_d \neq \emptyset$), and places the edit by

$$\text{place}(e) = \begin{cases} r & \text{if } \mathcal{R}(e) = \emptyset \text{ and } \Delta(e) > 0 \quad \text{PROMOTE} \\ \{\ell_d\}_{d \in \mathcal{B}(e)} & \text{if } \mathcal{B}(e) \neq \emptyset \text{ and } \mathcal{R}(e) \neq \emptyset \quad \text{LOCALIZE} \\ \perp & \text{if } \mathcal{B}(e) = \emptyset \quad \text{DISCARD} \end{cases} \tag{3}$$

where $\perp$ places nothing. An idealized characterization under exact scores and faithful localization is given in Proposition 1. The promote/localize distinction is read from executions the gate has already

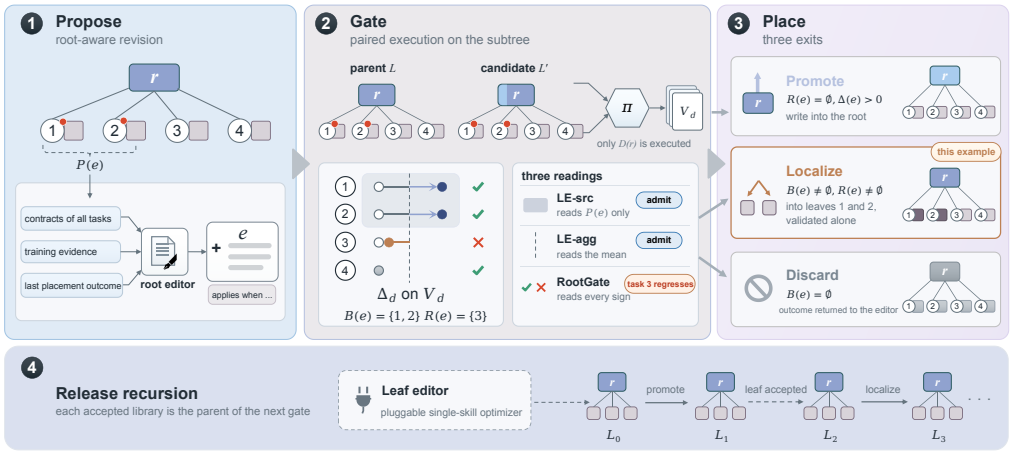

Figure 2: Overview of SCOPEEVO on a root r with four descendants. (1) A root editor proposes a candidate edit e from the training evidence of the whole subtree. (2) ROOTGATE executes the parent L and the candidate L' on the validation set of every descendant and reads the sign of each $\Delta_d(e)$. (3) The placement rule promotes, localizes, or discards e; here task 3 regresses, so e is localized into the leaves of tasks 1 and 2, whereas LE-SRC and LE-AGG would admit it into the root. (4) Every accepted library becomes the parent of the next event.

paid for, so localization adds no root executions. Three details complete the rule. Margins are fixed in advance: a number of net inputs on item-averaged descendants, and at least two fewer passes in five to seven paired repeats under the CoEvoSkills protocol; a candidate that passes on the validation sets is discarded if the descendants move in opposite directions on the training sets (Appendix B). A descendant that returns no usable score after a retry yields UNDECIDED and the parent stays, since zero-filling or dropping it would let one failed run decide a root. A root whose descendants are all at ceiling on their validation sets yields NOHEADROOM and is not edited.

Localization. A refused root edit usually contains correct knowledge about the descendants that improved. For each $d \in \mathcal{B}(e)$ the localize editor receives the candidate's diff and change notes together with d's contract and evidence, and rewrites the part relevant to d into ℓ_d as task-specific conventions that override the root where they conflict, without new failure reflection. The leaf candidate is accepted iff

$$J_d\big(L[\ell_d \leftarrow \ell'_d];\, V_d\big) \;>\; J_d(L;\, V_d) + \tau_d, \tag{4}$$

where $\tau_d \geq 0$ is a margin fixed in advance, zero for run-averaged descendants and a fixed number of items for item-averaged ones. By Corollary 1 a leaf edit leaves the score distributions of other tasks unchanged, so the criterion is evaluated on d alone: a root edit must be executed on the whole subtree, while its localized version needs one validation set and one editor call. A localization that fails this criterion leaves the library exactly as a discard would.

Leaf events. Independently of root events, a leaf editor may revise the leaf of a descendant that still fails on its training or validation set, and the revision is accepted by the same criterion. SCOPEEVO treats the leaf editor as a pluggable single-skill optimizer, since neither ROOTGATE nor the placement rule depends on its internals. Leaf events alone constitute the Local-only variant of our experiments, and their validation never needs the subtree.

5 ANALYSIS

The promote condition, $\mathcal{R}(e) = \emptyset$, asserts that no descendant regresses, on the strength of a few executions of a stochastic policy. We state what that assertion means. Let a run-averaged descendant d pass a validation run with probability p_d under the parent and p'_d under the candidate; we analyze the zero-margin gate with a fixed budget of n_d paired runs on V_d, which returns $d \in \mathcal{R}(e)$ iff $\Delta_d(e) < 0$. A regression, $p'_d < p_d$, is *missed* when $d \notin \mathcal{R}(e)$. Proofs, the scope lemma behind the gate, and the release-chain and false-rejection results are in Appendix C.

Theorem 1 (Detection limits of ROOTGATE)**.** *Fix a root edit e and a descendant d.*

(a) Deterministic conflict. *If $p_d > p'_d = 0$ and the parent's runs are independent, the miss probability is exactly $(1 - p_d)^{n_d}$, whatever the dependence between a parent run and its paired candidate run. For $p_d \in (0, 1)$, a miss probability of at most α needs $n_d \geq \log \alpha / \log(1 - p_d)$, which grows without bound as $p_d \to 0$.*

(b) Stochastic degradation. *If $p'_d = p_d - \delta$ with $0 < \delta < p_d$ and runs independent both within and across pairs, the miss probability is at least $\left(1 - p_d(1 - p'_d)\right)^{n_d}$; for $n_d = 2$, $p_d = 0.5$, and $p'_d = 0.3$ it exceeds 0.42.*

(c) Coverage. *Whatever the edit does to d on inputs outside V_d, if $n_d = 0$ or every input in V_d passes with probability one under both libraries, then $d \notin \mathcal{R}(e)$ almost surely.*

Theorem 1 says what ROOTGATE is: a detector of regressions on the descendants its validation sets cover, and not a statistical test of improvement. Parts (a) and (b) are its sampling limit: with a few runs a stochastic degradation is often missed, and a descendant that rarely passes under the parent is the hardest to protect, but for a regression of fixed size the miss probability decays exponentially in n_d. Part (c) is its coverage limit, which no number of runs removes. Two design choices follow. A missing score is neither zero-filled nor dropped, because the former would count as a detection and the latter as a pass. A root whose descendants are all at ceiling is not edited, because no candidate could show the improvement a promotion requires. The theorem also bounds what the gate can promise: along the release chain, each accepted event passes the gate's own paired check against its parent, and by part (c) its protection does not reach inputs that its validation sets do not exercise.

6 EXPERIMENTAL RESULTS

6.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on two benchmarks, each organized around shared roots. On question answering we use SearchQA and DocVQA with the splits and scorers of Yang et al. (2026c), exact match and ANLS respectively; the two domains are the two descendants of one shared root. On SkillsBench (Li et al., 2026b) we use the tasks whose skills are shared with other tasks, since only single-skill optimization applies to the rest, and follow the protocol of Zhang et al. (2026b): a task is scored by its deterministic verifier and passes only when all of its tests pass. On both benchmarks we build the hierarchy by delivering to each task a shared root and a task-specific leaf (Appendix B).

Evaluation. On QA, editors see execution evidence from the training set only, ROOTGATE decides placement and leaf acceptance on the validation set, and the test set is scored once on final libraries. On SkillsBench, as in CoEvoSkills, the task's own instance serves both the loop and the evaluation, and the final library is scored by five fresh runs per task. We report J, the equal-weight mean of per-descendant scores on the stated set, and beside each gate decision the number of inputs that improve and regress relative to the parent library, since a descendant whose mean rises can still regress on individual inputs. Splits, prompts, and acceptance thresholds were fixed before each stage, and test sets were scored only after every final library was frozen (Appendix B.5).

Implementation. The QA executor is Qwen3.6-35B-A3B, and the SkillsBench executor is Claude Opus 4.6 through Claude Code in the SkillsBench harness; executors, decoding configurations, and scorers are fixed across conditions. Each editor call may change a bounded amount of text, and each benchmark has a fixed budget of root and leaf events, which SCOPEEVO may spend root-first or leaf-first (Appendices B and D.1). The SkillsBench scope ablation (Figure 5) and the transfer experiment run on a separate placement testbed with unseen instances, described in Appendix B.3.

Baselines. On SkillsBench we compare against no skill, the human-curated skills released with the benchmark (Li et al., 2026b), and CoEvoSkills (Zhang et al., 2026b). On QA we compare against SkillOpt (Yang et al., 2026c) and the skill baselines it reports: human-written and LLM-written skills, Trace2Skill (Ni et al., 2026), and the prompt optimizers TextGrad (Yuksekgonul et al., 2024) and GEPA (Agrawal et al., 2026). To isolate the effect of placement, we also run two single-scope variants of SCOPEEVO from the same initial library: Root-only, which edits only roots and can promote or discard a candidate but not localize it, and Local-only, which freezes the roots and edits only leaves, and so cannot reach a task it does not observe.

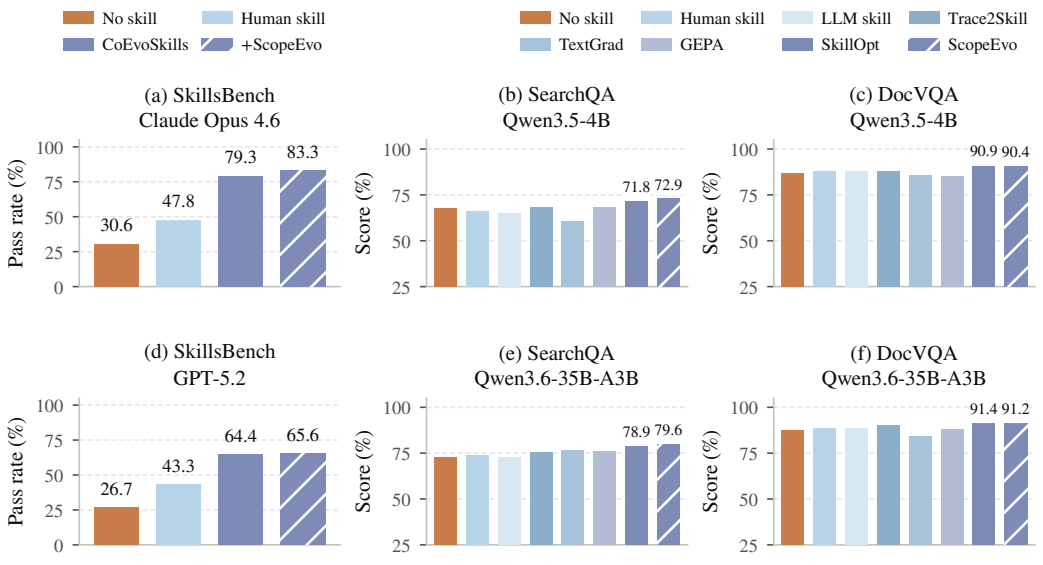

Figure 3: Main results. (a, d) Pass rate (%) on the 30 shared-skill SkillsBench tasks under the CoEvoSkills protocol, with Claude Opus 4.6 via Claude Code (a) and GPT-5.2 via Codex (d); the library evolved in (a) is evaluated in (d) without further evolution. (b, c, e, f) Test scores (%) on SearchQA and DocVQA under SkillOpt's harness and metrics, with Qwen3.5-4B (b, c) and Qwen3.6-35B-A3B (e, f) as executors; SCOPEEVO and our SkillOpt run are optimized with Qwen3.6-35B-A3B. Other QA scores are as reported by Yang et al. (2026c).

6.2 MAIN RESULTS

Results on SkillsBench. We first ask whether scoped root evolution adds to a library that a strong library-level method has already evolved. Starting from the CoEvoSkills library, SCOPEEVO evolves the shared root of each skill family and the leaves beneath it, and Figure 3(a, d) reports the pass rate on the 30 shared-skill tasks. SCOPEEVO raises the pass rate of CoEvoSkills from 79.3% to 83.3%, and no task falls. Its leaf editor runs CoEvoSkills' own reflection-and-validation protocol, and ROOTGATE places every root candidate on top of it. We then ask whether the evolved skills are tied to the executor that produced them. The library optimized under Claude Opus 4.6 is mounted unchanged on GPT-5.2 through Codex, with no further evolution. It improves the pass rate there as well (64.4% to 65.6%), which is descriptive evidence of transfer across executors in this setting.

Results on QA tasks. On SearchQA and DocVQA, the two descendants of one shared root, we compare against SkillOpt and the skill baselines it reports (Figure 3(b, c, e, f)). With the Qwen3.6-35B-A3B executor, SCOPEEVO is on par with SkillOpt, the strongest single-skill optimizer, on both domains and ahead of every other skill. This is the outcome the subtree predicts rather than a ceiling on the method: the two QA domains share little at the rule level (Appendix D.3), so most of what there is to learn is task-specific and lives in the leaves, where SCOPEEVO's leaf editor follows SkillOpt's own protocol. The comparison thus shows compatibility. Where the descendants of a root share nothing, SCOPEEVO reduces to the single-skill optimizer it wraps and loses nothing to it; where they share something, root placement is available in addition. Evaluated unchanged on the smaller Qwen3.5-4B executor, the same library keeps its lead on SearchQA, so the rules it learned are not specific to the executor they were written under.

Beyond the pass rate. A mean score shows how often a library succeeds, but not how consistently it does so or at what optimization cost. We therefore ask two further questions (Figure 4(a, b)): whether SCOPEEVO's gain on SkillsBench holds across repeated runs, and whether its placement layer adds cost on top of the single-skill optimizer it wraps. On SkillsBench with Claude Opus 4.6, the gain is one of reliability. Under pass^k (Yao et al., 2024), which counts a task only if all k of its runs pass, the gap to CoEvoSkills widens from 4.0 points at $k{=}1$ to 6.7 points at $k{=}5$, while

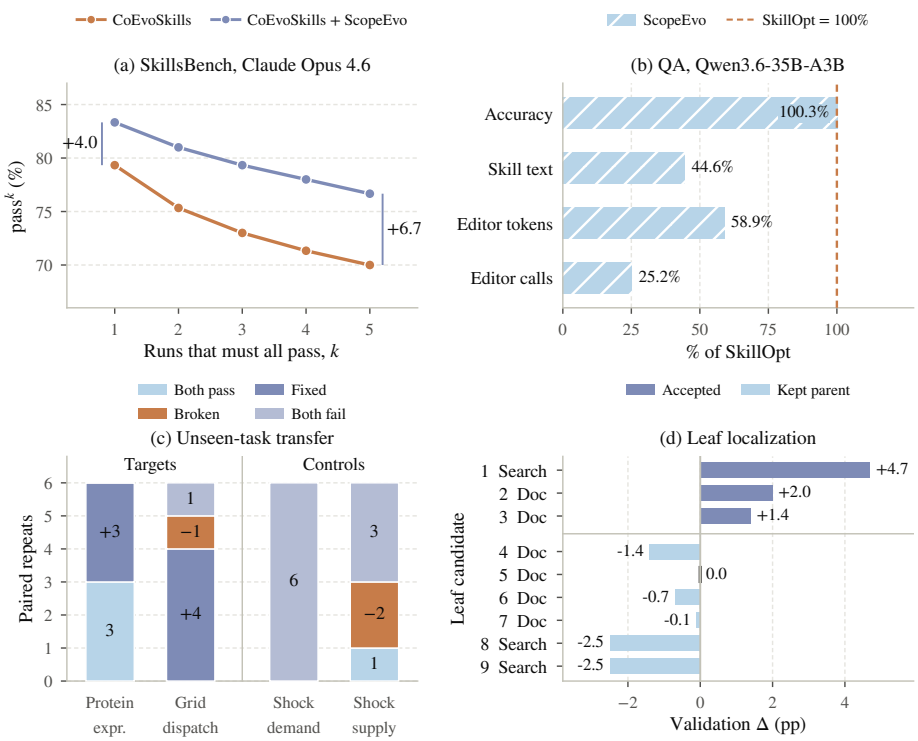

Figure 4: Beyond the mean pass rate (a, b) and where edits take effect (c, d). (a) pass^k (Yao et al., 2024) on SkillsBench; k=1 is the pass rate of Figure 3(a). (b) SCOPEEVO relative to our SkillOpt run on QA in accuracy, skill text in the executor's context, and optimization tokens and calls. (c) Four SkillsBench tasks held out from every editor input and validation run, under the initial library and Root-only; two are controls whose failure no validation instance exercises. (d) Leaf candidates 1–9: root candidates refused by ROOTGATE and rewritten into the leaf of the descendant they helped (Search: SearchQA, Doc: DocVQA), with their validation Δ (pp) on that descendant.

the share of tasks passed in at least one of the five runs stays at 86.7% for both. SCOPEEVO thus turns intermittent passes into consistent ones rather than solving tasks that were never solved. The promoted roots are in active use: the executor reads them in 104 of the 115 runs that mount them. They also add no partial regressions: on the 18 tasks whose skills changed, the number of passed tests rose on four and fell on none, consistent with the gate's no-regression rule. On QA, SCOPEEVO costs less in total: it matches SkillOpt while using 25% of the calls and 59% of the tokens that SkillOpt spends on optimization and delivering 45% as much skill text to the executor. The more compact library does not give up answer quality, since its SearchQA F1 is the highest among our runs on both executors. Where descendants share a root, placement therefore adds reliability; where they share little, SCOPEEVO costs less in total than the optimizer it wraps.

6.3 EFFECT ANALYSIS

Shared roots can carry one repair to many descendants. Only a root edit can reach a descendant that did not propose it, so we test whether a promoted root repairs tasks that played no part in producing it. In the transfer run, four descendants of the spreadsheet and power-flow roots were held out from every editor input and validation run, and were executed on their own instances only after both roots had been evolved on the remaining descendants and frozen (Figure 4(c)). Two are transfer targets; the other two are controls whose failures stem from an unstated unit convention that no validation instance exercises. Each family's root repaired its transfer target: protein-expression-analysis rose from 3/6 to 6/6 with no regression and grid-dispatch-operator from 1/6 to 4/6, whereas Local-

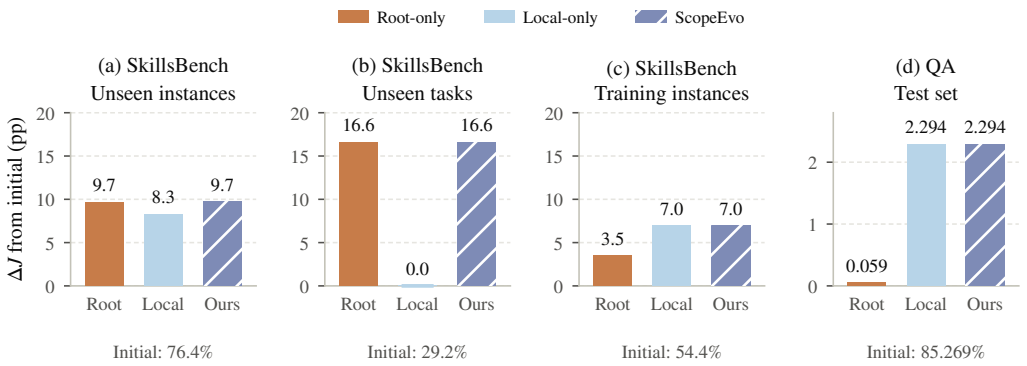

Figure 5: scope ablation: change in J (pp) from the initial library for Root-only, Local-only, and SCOPEEVO. SkillsBench unseen instances (a), unseen tasks (b), and the QA test set (d) measure generalization; SkillsBench training instances (c) were visible to the editors and measure repair of shown failures. Panel (a) covers 13 tasks with an unseen instance and five reference tasks whose library no condition changed, scored on previously used inputs. Panel (d) has its own scale.

only leaves all four tasks at the initial library by construction. A shared cause is thus diagnosed once at the root, repaired once, and inherited by descendants the loop never diagnosed.

Rejected root edits can be recovered in the leaves. On the QA subtree, most root candidates raise one descendant and lower its sibling on the validation set (Appendix D.3); a gate with only two exits would discard them, including those that hold correct knowledge at the wrong level. The localize exit instead rewrites such a candidate into the leaf of the descendant it helped. To test whether the relocated knowledge holds beyond the validation set, we compare pairs of libraries that differ in exactly one localized leaf (Figure 4(d)): the hand-written root with and without leaf candidate 1, the SearchQA rewrite of a candidate refused for lowering DocVQA, and SCOPEEVO under the root-first schedule against Root-only, which differ only in leaf candidate 3, the DocVQA rewrite of another refused candidate. In both pairs the leaf's domain improves on the test set while the other domain is unchanged by construction, and leaf candidate 1 raises J by $+0.786$ pp.

Scope ablation. To separate where an edit is placed from whether it is made, we compare SCOPEEVO with Root-only and Local-only from the same initial library on both testbeds (Figure 5). Root placement wins where descendants share a cause: on SkillsBench unseen instances and on the unseen tasks, which Local-only cannot reach at any budget. Leaf placement wins where failures are task-specific: on SkillsBench training instances and on the QA test set, where 11 of the 16 root candidates move SearchQA and DocVQA in opposite directions on the validation set (Appendix D.3), a pattern consistent with knowledge that holds for one descendant and not for its sibling. Neither scope alone covers both cases. ROOTGATE places each candidate by the observed signs, promoting it when no descendant falls and localizing it when one rises and another falls, so that SCOPEEVO obtains the transfer of the root scope and the task-specific gains of the leaf scope from one loop.

7 CONCLUSION

Root skills are where a hierarchical library can transfer, but their broad scope also makes them dangerous to edit. We turn this risk into a placement decision: an edit belongs in the root only if no evaluated descendant regresses under paired execution, and otherwise in the leaves of the descendants it helps. SCOPEEVO makes this decision from the gate's own executions and complements existing methods by further improving their libraries; promoted roots repair unseen tasks, while refused root edits are recovered rather than discarded. For a self-evolving library, where an edit is written thus matters as much as what it says. What the validation set does not cover, the gate cannot protect, motivating stronger validation, deeper placement, and transfer across roots.

AI USE STATEMENT

We used generative AI tools to assist with drafting sections of the manuscript, refining and polishing the writing, and retrieving and discovering relevant literature. We also used generative AI tools to assist with writing and organizing the code. All AI-assisted content was reviewed and revised by the authors. We take responsibility for the final content of this work, including its text, references, claims, and any artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Appendix A provides the pseudocode for SCOPEEVO. Appendix B describes the datasets and splits, scoring procedures, library construction and skill delivery, executor configurations, editing budgets, and acceptance thresholds; Appendix B.5 further specifies the evaluation protocol. Appendix C states the theoretical assumptions and provides the proofs, while Appendix D reports additional condition-level and task-level results. The supplementary material includes anonymized core algorithm code to support inspection of the implementation and its correspondence to the method described in the paper.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Zihan Chen, Songwei Dong, Chengshuai Shi, Peng Wang, Song Wang, Cong Shen, and Jundong Li. The past is prologue: A plug-in controller for selective updates in sequentially evolving llm memory. *arXiv preprint arXiv:2606.31121*, 2026.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *Advances in Neural Information Processing Systems*, 37:71596–71642, 2024.

Mohd Sameen Chishti, Damilare Peter Oyinloye, and Jingyue Li. Test before you deploy: Governing updates in the llm supply chain. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pp. 1584–1587, 2026.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024.

Yu He and Weikai Yang. Skillcommit: Evolving agent skills through behaviorally validated scope expansion. *arXiv preprint arXiv:2608.15165*, 2026.

Alejandro Breen Herrera, Aayush Sheth, Steven G Xu, Zhucheng Zhan, Charles Wright, Marcus Yearwood, Hongtai Wei, Sudeep Das, Danny Nightingale, Meg Watson, et al. Build, judge, optimize: A blueprint for continuous improvement of multi-agent consumer assistants. *arXiv preprint arXiv:2603.03565*, 2026.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Hao Li, Chunjiang Mu, Jianhao Chen, Siyue Ren, Zhiyao Cui, Yiqun Zhang, Lei Bai, and Shuyue Hu. Organizing, orchestrating, and benchmarking agent skills at ecosystem scale. *arXiv preprint arXiv:2603.02176*, 2026a.

Jialong Li and Jialing Zhu. Auditing self-evolution in financial agents: Capability gains, security drift, and execution-interface mismatch. *arXiv preprint arXiv:2608.17684*, 2026.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026b.

Yubo Li. Dynamic agent skills: A lifecycle survey and taxonomy of evolving skill libraries. *arXiv preprint arXiv:2607.10113*, 2026.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*, 2026.

Johannes Moll, Jean-Philippe Corbeil, Jiazhen Pan, Martin Hadamitzky, Daniel Rueckert, Lisa Adams, and Keno Bressem. Grasp: Gated regression-aware skill proposer for self-improving llm agents. *arXiv preprint arXiv:2605.29668*, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, et al. Skillos: Learning skill curation for self-evolving agents. *arXiv preprint arXiv:2605.06614*, 2026.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 7957–7968, 2023.

Hongji Pu, Xinyuan Song, and Liang Zhao. Skillops: Managing llm agent skill libraries as self-maintaining software ecosystems. *arXiv preprint arXiv:2605.13716*, 2026.

Libin Qiu, Zhirong Gao, Junfu Chen, Yuhang Ye, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Shuo Tang. Autorefine: From trajectories to reusable expertise for continual llm agent refinement. *arXiv preprint arXiv:2601.22758*, 2026.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Sina Tayebati, Divake Kumar, Nastaran Darabi, Ranganath Krishnan, and Amit Ranjan Trivedi. Self-healing harness for runtime oversight of agent self-modification, 2026. URL `https://arxiv.org/abs/2609.24130`.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Yuan Xiong, Ziqi Miao, Qian Chen, Lijun Li, Yequan Wang, Shizhu He, Jun Zhao, and Kang Liu. Skillpyramid: A hierarchical skill consolidation framework for self-evolving agents. *arXiv preprint arXiv:2606.03692*, 2026.

Haoran Xu, Jiacong Hu, Ke Zhang, Lei Yu, Yuxin Tang, Xinyuan Song, Yiqun Duan, Lynn Ai, and Bill Shi. Sedm: Scalable self-evolving distributed memory for agents. *arXiv preprint arXiv:2509.09498*, 2025.

Chen Yang, Jiashuo Tian, Ziqi Wang, Xinyin Liu, Meiru Ye, and Junjie Chen. Learning globally reusable skills for coding agents. *arXiv preprint arXiv:2608.06153*, 2026a.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Min Yang, Jinghua Piao, Xu Xia, Xiaochong Lan, Jiaju Chen, Yongshun Gong, and Yong Li. Skillmaster: Toward autonomous skill mastery in llm agents. *arXiv preprint arXiv:2605.08693*, 2026b.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026c.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL `https://arxiv.org/abs/2406.12045`.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023.

Di Zhang. Agentdevel: Reframing self-evolving llm agents as release engineering. *arXiv preprint arXiv:2601.04620*, 2026.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026b.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026c.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Lechen Zhang, Tolga Ergen, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. Sprig: Improving large language model performance by system prompt optimization. In *International Conference on Learning Representations*, volume 2026, pp. 26649–26672, 2026d.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026e.

Ziao Zhang, Kou Shi, Shiting Huang, Avery Nie, Yu Zeng, Yiming Zhao, Zhen Fang, Qishen Su, Haibo Qiu, Wei Yang, et al. Skillflow: Benchmarking lifelong skill discovery and evolution for autonomous agents. *arXiv preprint arXiv:2604.17308*, 2026f.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, et al. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*, 2026.

Algorithm 1 SCOPEEVO: one root event followed by leaf events, on library L_k

Root event for a root r with attributable failures on its training sets; NOHEADROOM if every $d \in \mathcal{D}(r)$ is at ceiling on V_d
  $e \leftarrow$ ROOTEDITOR$\big(r,\ \{\text{contract, training evidence}\}_{d\in\mathcal{D}(r)},\ \text{previous placement outcome}\big)$, or NOCHANGE
  paired execution of L_k and $L_k[r \leftarrow r \oplus e]$ on V_d for every $d \in \mathcal{D}(r)$; $\mathcal{B} \leftarrow \{d : \Delta_d(e) > 0\}$, $\mathcal{R} \leftarrow \{d : \Delta_d(e) < 0\}$
  if some d returns no usable score: UNDECIDED, keep L_k
  elif $\mathcal{R} = \emptyset$ and $\Delta(e) > 0$ and the training-set check passes: PROMOTE, $L_{k+1} \leftarrow L_k[r \leftarrow r \oplus e]$
  elif $\mathcal{B} \neq \emptyset$ and $\mathcal{R} \neq \emptyset$: LOCALIZE: for each $d \in \mathcal{B}$, $\ell'_d \leftarrow$ LOCALIZEEDITOR(e, ℓ_d), accepted iff equation 4; root unchanged
  else DISCARD; return the paired outcome to the root editor
Leaf events for each d with residual failures: $\ell'_d \leftarrow$ LEAFEDITOR$(\ell_d,\ \text{training evidence of } d)$, accepted iff equation 4
Release recursion: the accepted library becomes the parent of the next event

A THE SCOPEEVO ALGORITHM

Algorithm 1 states one root event followed by leaf events on a library L_k; Figure 2 illustrates it on one root with four descendants.

B DATASETS, TESTBEDS, AND EVALUATION PROTOCOL

Every number in this paper comes from one of four settings: the two official settings of §6.1 (Figure 3) and the two placement testbeds on which the ablations run (Figure 5, Figure 4(c, d), and Appendix D). Each table names its setting.

B.1 QA: THE SKILLOPT SETTING AND THE QA SUBTREE

Data and scorers. SearchQA and DocVQA follow the released split of Yang et al. (2026c): $400/200/1{,}400$ SearchQA and $107/53/374$ DocVQA questions for training, validation, and test. The training set supplies the executions from which editors take evidence; the validation set is the set on which SkillOpt itself selects skills, and it is where ROOTGATE runs and leaf edits are accepted; the test set is scored only on final libraries. SearchQA is scored by exact match and DocVQA by ANLS. Figure 3(b, c, e, f) reports SkillOpt's native scores from its own harness; the ablations run in our harness and score DocVQA by mean ANLS, the graded per-item score the gate reads, so that a gate decision is reproducible item by item and per-item improvement and regression counts are meaningful. The two are never mixed.

The official QA results. For Figure 3(b, c, e, f), SCOPEEVO's final QA library and our SkillOpt skills are evaluated in SkillOpt's released harness with its default decoding, one run per library and executor, with the root and the domain's leaf filling the harness's skill slot. Our SkillOpt row runs SkillOpt's released optimizer with the same executor. Both skill sets were optimized under the Qwen3.6-35B-A3B executor and are evaluated unchanged on Qwen3.5-4B. Figure 4(b) compares the two optimization runs as wholes, so it measures total cost and does not isolate the cost of the placement layer.

The QA subtree. The library is one root and two leaves. The system prompt opens with a fixed role sentence, then the root inside a `shared_qa_guidance` block, then the domain's leaf inside a `task_guidance` block, then the frozen task protocol. A leaf is delivered only to its own domain, an empty node is delivered as a placeholder rather than an omitted block, and the precedence sentence is fixed: "Apply the shared guidance first. Where the task guidance states a more specific rule for this task, the task guidance takes precedence." The initial library is empty. The executor is Qwen3.6-35B-A3B (6-bit, served locally) at temperature 0 with thinking disabled, 2,048 generation tokens over at most four turns and three tool actions; tools reach only the current question's evidence (text search and a windowed read for SearchQA, image viewing for DocVQA).

Budgets and bounds. Each condition has a fixed number of editor opportunities: Local-only has 8 SearchQA and 4 DocVQA leaf events, Root-only has 10 root events, and SCOPEEVO has the allocations of Figure 6; a branch stops after five consecutive refusals. An editor call proposes at most four edits with a net growth of at most 500 tokens; the root is at most 1,000 tokens, and the root and leaf a task receives together are at most 1,500 tokens, the size of SkillOpt's largest released skill. Editors see the training set only, as a packet of at most six failed and four successful executions per domain selected by failure type, and never see validation or test items or scores. After a discard or a promotion, the root editor receives only the placement label; after a localization, it also receives the domain that benefited, which reflects the direction of the validation outcome but carries no score or item.

Acceptance thresholds, fixed in advance. A SearchQA leaf is accepted iff it gains at least two net items on the 200 validation items and at least two on the 400 training items; a DocVQA leaf iff it gains at least $1.0\,\mathrm{pp}$ mean ANLS on the 53 validation items and $0.5\,\mathrm{pp}$ on the 107 training items. A root candidate is promoted iff neither domain falls and their mean rises on the validation set and again on the training set; it is localized iff one domain rises and the other falls on the validation set; otherwise it is discarded. Leaf candidate 1 of Figure 4(d) was validated on a 64-item subset of the SearchQA validation set; its test-set effect is measured, like leaf candidate 3's, as the paired difference of two libraries that differ only in that leaf. The hand-written root of §6.3 is a 252-token root written before any run and serves only as the reference library of that pair.

B.2 SKILLSBENCH: THE COEVOSKILLS SETTING

This is the setting of Figure 3(a, d).

Tasks and library. We use the SkillsBench tasks whose skills are shared with other tasks, in nine families; the remaining tasks each carry a skill of their own, so that only single-skill optimization applies to them, and they are not included. Figure 3(a, d) reports the 30 of them on which every row is scored under the same verifier. CoEvoSkills' published numbers average over all of its tasks and are not restricted to this subset. Each task's released CoEvoSkills package is its leaf; each family's shared package is its root, which starts empty and is written by the root editor.

Protocol. As in CoEvoSkills, the loop runs on the task's own instance. The editor receives executions of that instance (the trajectory, outputs, and the verifier's pass/fail outcome with its coarse categories, never the tests or expected values); ROOTGATE compares candidate and parent on paired repeats of the same instance, five per side, topped up to seven when the outcome is contested, counting a descendant as regressing when the candidate passes at least two fewer repeats, and promotes only if no descendant regresses and the family's net pass count rises; the final library is scored by five fresh runs per task on Claude Opus 4.6, a task passing only when all of its tests pass. The root editor is called once per family and receives no placement feedback. Two screens precede validation: candidate text is checked against the tasks' reference solutions and tests, a match rejecting the candidate, and a review step vetoes candidates that hard-code instance values, remove content the task requires, or modify the execution environment. Tasks whose library SCOPEEVO leaves unchanged share the baseline executions. The executor is Claude Opus 4.6 through Claude Code in the SkillsBench harness; for the GPT-5.2 rows the same libraries are mounted unchanged through Codex and scored by three runs per task. The No-skill and Human-skill rows are released SkillsBench runs on these tasks.

B.3 SKILLSBENCH: THE PLACEMENT TESTBED

This is the setting of the SkillsBench panels of Figure 5, of Figure 4(c), and of Appendix D.2.

Why a second testbed. The CoEvoSkills protocol scores a library on the instances the loop saw, so it measures repair of shown failures; it cannot separate root placement from leaf placement, and it cannot show transfer to inputs or tasks the loop never saw. The placement testbed adds both: every task carries instances the loop never sees, and one of its two runs sets aside whole tasks. It shares tasks with the main setting and nothing else; no library, execution, or editor output is reused between the two.

Conditions. Three conditions start from the same initial library: the initial library itself; Local-only, which freezes the roots and edits only leaves; and Root-only, which edits only roots and can

promote or discard a candidate but not localize it. Local-only and Root-only are the two single-scope ablations of SCOPEEVO, so together they separate the effect of where an edit is placed from the effect of editing at all. The same three conditions are run on the QA subtree, where SCOPEEVO's own runs, under both schedules, are reported in Appendix D.1.

Tasks and library. 21 tasks, under seven shared skill packages that serve as roots: five that the benchmark's tasks share, and two further packages, each read by a pair of tasks. The initial library is the benchmark's released skills, with the Three.js package carrying one revision common to every condition. A task's leaf is its own private package where it has one (nine tasks); the other twelve receive an empty leaf the first time an edit is localized to them. The executor is GPT-5.6 through the Codex agent, pinned by model, agent version, and benchmark commit. The agent reads mounted packages itself; the recorded trajectories show that promoted roots and accepted leaves were read.

Instance sets. Training instances are the tasks' official instances and are visible to editors. After a discard or a promotion, the root editor receives the candidate with fresh runs of it on these training instances, which is the paired outcome of §4; the gate's validation result reaches it only as the placement label. Validation instances are inputs generated for this study and fixed in advance; the gate runs on them, zero to four per task, and two tasks have none because their graders block input generation. Unseen instances are new inputs generated without any model, qualified against the task's oracle, frozen before the run that used them, and never shown to editor or gate: 13 tasks have one, five tasks are scored on inputs whose library no condition changed, and three have none. The two power-flow unseen instances were frozen before that family's root candidates were proposed, and their oracle qualification was checked afterwards, before any library was scored on them. Repeats are four for the three tasks whose library changed under more than one condition and two otherwise.

Two runs. In the full run all 21 tasks are visible to editors and gate; it yields the unseen-instance and training-instance panels of Figure 5 and the per-task tables of Appendix D.2. In the transfer run four tasks enter no editor input and no validation run, for reasons fixed in advance: two transfer targets, protein-expression-analysis, which is at ceiling on its validation runs, so that setting it aside costs the gate nothing while its own instances have headroom, and grid-dispatch-operator, which makes the power-flow family a transfer test at the cost of the gate's second descendant; and two controls, shock-analysis-demand and shock-analysis-supply, which have no validation instance and fail for an unstated unit convention, so that no repair was expected of them. Both roots were evolved on the remaining descendants and frozen; only then were the four unseen tasks executed, six paired repeats each (Figure 4(c)). In each run four root candidates were decided, two promoted and two discarded.

B.4 SCORES AND INTERVALS

On QA, a descendant's score is its mean over the stated set (exact match or ANLS) and J is the unweighted mean of the two domains, so the 374 DocVQA items carry the same weight as the 1,400 SearchQA items. Intervals are a stratified paired bootstrap against the starting library, 10,000 draws with a fixed seed, stratified by domain and clustered by SearchQA near-duplicate group and by DocVQA document. On SkillsBench a descendant's score is its pass rate over repeats and J is the unweighted mean over the tasks with the stated input type; we report counts rather than intervals there, because six paired repeats separate $1/6$ from $5/6$ but do not license a significance statement. Scoring is programmatic on both testbeds – verifiers, exact match, ANLS – and no LLM judge is used anywhere in this paper.

B.5 EVALUATION PROTOCOL

Three rules hold on the QA subtree and on the placement testbed. Editors receive training-set evidence only, and we checked mechanically that no editor input contains a validation, test, or unseen-task item; on QA the check covers all 252 editor requests, none of which contains a validation score either. The placement label is thus an editor's only, and coarse, signal from the validation set. The validation set is read only by ROOTGATE and by leaf acceptance, with thresholds fixed before each stage. Test sets and unseen tasks are scored only on frozen final libraries. Under the CoEvoSkills protocol each task has a single instance, which the loop and the evaluation share, as they do for CoEvoSkills' published numbers; the leaf editor, following that protocol, sees the runs of its rejected drafts on that instance. Figure 3(a, d) therefore measures repair of the benchmark's tasks; generalization to inputs and tasks the loop never saw is measured on the placement testbed (Figure 5 and

Figure 4(c)) and on the QA test set. On the placement testbed the training instances are the tasks' official instances, the same kind of evaluation as the official protocol; we report them separately from unseen instances, and sets are never averaged.

C THEORETICAL DETAILS AND PROOFS

This appendix states the assumptions behind the analysis, proves the scope lemma, characterizes idealized placement, and proves Theorem 1 together with the sampling, release-chain, and false-rejection results.

C.1 SETTING AND ASSUMPTIONS

We use the notation of §3–5 and three assumptions.

(A1) *Tree.* Each task d reads one root $r(d)$ and one leaf ℓ_d, and ℓ_d is read by d alone; thus $\mathcal{D}(r) = \{d : r(d) = r\}$.

(A2) *Context sufficiency.* For every input x, the distribution of $s_d(L, x)$ depends on L only through $c_d(L) = \phi(r(d), \ell_d)$, and every execution draws fresh randomness from π.

(A3) *Run model.* For a run-averaged descendant, the gate observes n_d paired runs (X_j, X'_j), $j = 1, \dots, n_d$, the pass indicators under the parent and the candidate. Given the candidate and the history of the loop, the pairs are i.i.d. across j with $X_j \sim \mathrm{Bern}(p_d)$ and $X'_j \sim \mathrm{Bern}(p'_d)$; within a pair any dependence is allowed unless stated otherwise.

We write $\bar{J}_d(L; X)$ for the expectation of $J_d(L; X)$ over the policy's randomness and $\bar{\Delta}_d(e) = \bar{J}_d(L') - \bar{J}_d(L)$; under (A3), $\bar{J}_d(L; V_d) = p_d$ and $\bar{J}_d(L'; V_d) = p'_d$. The gate returns $d \in \mathcal{R}(e)$ iff the observed $\Delta_d(e)$ is negative, and a regression on V_d, $p'_d < p_d$, is *missed* if $d \notin \mathcal{R}(e)$.

C.2 SCOPE AND VALIDATION ASYMMETRY

Lemma 1 (Scope)**.** *Under (A1), an edit to root r leaves c_d unchanged for every task $d \notin \mathcal{D}(r)$, and an edit to leaf ℓ_d leaves $c_{d'}$ unchanged for every task $d' \neq d$. If ϕ is injective in the edited argument, the edit changes c_d for every $d \in \mathcal{D}(r)$, respectively for d.*

Proof. By (A1), $c_d(L) = \phi(r(d), \ell_d)$ depends on L only through these two nodes. An edit to r changes the first argument exactly for the tasks with $r(d) = r$, which form $\mathcal{D}(r)$, and changes no second argument; an edit to ℓ_d changes the second argument of d only. Injectivity turns a changed argument into a changed context. □

Corollary 1 (Validation asymmetry)**.** *Call a validation procedure* sound *for an edit if it executes every task whose score distribution the edit can change. Under (A1)–(A2), executing $\mathcal{D}(r)$ is sound for a root edit, and executing d is sound for an edit to ℓ_d.*

Proof. By (A2), a task's score distribution can change only if its context does, and by Lemma 1 the contexts that can change are those of $\mathcal{D}(r)$, or of d alone. □

C.3 IDEALIZED PLACEMENT

Proposition 1 (Idealized placement characterization)**.** *Suppose every descendant is evaluated, the gate observes $\bar{\Delta}_d(e)$ exactly, and localization is* faithful*: for every $S \subseteq \mathcal{D}(r)$, writing e into the leaves of S changes $\bar{J}_d$ by $\bar{\Delta}_d(e)$ for $d \in S$ and leaves every other descendant unchanged. Then the placement returned by equation 3 attains the maximum of equation 2. When $\mathcal{R}(e) = \emptyset$ and $\mathcal{B}(e) \neq \emptyset$, root placement and leaf placement on $\mathcal{B}(e)$ attain the same value, and equation 3 chooses the root.*

Proof. Let $m = |\mathcal{D}(r)|$. Root placement has value $\frac{1}{m}\sum_d \bar{\Delta}_d(e)$ and is feasible iff $\mathcal{R}(e) = \emptyset$. Leaf placement on S has value $\frac{1}{m}\sum_{d \in S} \bar{\Delta}_d(e)$ and is feasible iff $S \cap \mathcal{R}(e) = \emptyset$; among feasible S the value is maximized by $S = \mathcal{B}(e)$, because descendants outside $\mathcal{B}(e) \cup \mathcal{R}(e)$ contribute zero, and

$S = \emptyset$, a discard, has value zero. If $\mathcal{R}(e) = \emptyset$ and $\mathcal{B}(e) \neq \emptyset$, then $\Delta(e) > 0$, and root placement and leaf placement on $\mathcal{B}(e)$ both attain $\frac{1}{m}\sum_{d\in\mathcal{B}(e)} \bar{\Delta}_d(e) > 0$, the maximum; equation 3 promotes. If $\mathcal{R}(e) \neq \emptyset$ and $\mathcal{B}(e) \neq \emptyset$, root placement is infeasible and $\mathcal{B}(e)$ attains the maximum; equation 3 localizes to $\mathcal{B}(e)$. If $\mathcal{B}(e) = \emptyset$, every feasible placement has value zero, which a discard attains; equation 3 discards, and ties keep the parent. □

This characterization assumes faithful localization. In the method, each rewritten leaf is evaluated separately by equation 4.

C.4 Proof of Theorem 1 and the sampling limit

Proof of Theorem 1. Under (A3), for $n_d \geq 1$ let $D_j = X'_j - X_j$, so that $\Delta_d(e) = \frac{1}{n_d}\sum_j D_j$ and $d \in \mathcal{R}(e)$ iff $S = \sum_{j=1}^{n_d} D_j < 0$; a miss is the event $\{S \geq 0\}$.

(a) If $p'_d = 0$, then $X'_j = 0$ almost surely and $S = -\sum_j X_j$, so $\{S \geq 0\} = \{X_j = 0 \text{ for all } j\}$. This event involves the parent runs only, so the dependence within a pair is irrelevant, and since the parent runs are independent its probability is $(1-p_d)^{n_d}$. For $p_d \in (0,1)$, $(1-p_d)^{n_d} \leq \alpha$ iff $n_d \geq \log\alpha/\log(1-p_d)$, whose right-hand side grows without bound as $p_d \to 0$. If the runs are independent but not identically distributed, as when they are spread over several inputs with pass probabilities $p_{d,j}$, the miss probability is $\prod_j(1-p_{d,j}) \leq (1-\bar{p}_d)^{n_d}$, with $\bar{p}_d$ their mean, by the inequality of arithmetic and geometric means.

(b) The miss event contains the event that no pair equals $(1,0)$, since then every $D_j \geq 0$. With i.i.d. pairs, this event has probability $\big(1-\Pr[(X_j,X'_j)=(1,0)]\big)^{n_d}$. If X_j and X'_j are independent, $\Pr[(1,0)] = p_d(1-p'_d)$, which gives the bound; with $n_d = 2$, $p_d = 0.5$, and $p'_d = 0.3$ it equals $0.65^2 = 0.4225$. The bound also holds whenever the covariance within a pair is non-negative, because then $\Pr[(1,0)] = p_d - \Pr[(1,1)] \leq p_d - p_d p'_d$.

(c) If $n_d = 0$, the gate has no run of d, so d enters neither $\mathcal{B}(e)$ nor $\mathcal{R}(e)$. If every input in V_d passes with probability one under both libraries, then $X_j = X'_j = 1$ almost surely, $S = 0$, and $d \notin \mathcal{R}(e)$. Both statements depend only on executions on V_d, so they hold whatever the edit does on other inputs. □

Proposition 2 (The sampling limit vanishes)**.** *Under (A3), if $p'_d \leq p_d - \delta$ with $\delta > 0$, the miss probability is at most $\exp(-n_d\delta^2/2)$, whatever the dependence within a pair.*

Proof. The D_j are i.i.d., take values in $[-1,1]$, and have mean $p'_d - p_d \leq -\delta$. By Hoeffding's inequality, $\Pr[S \geq 0] \leq \Pr[S - \mathrm{E}S \geq n_d\delta] \leq \exp\big(-2(n_d\delta)^2/(4n_d)\big) = \exp(-n_d\delta^2/2)$. □

The bounds in Theorem 1, Proposition 2, and Proposition 3(ii) concern a gate with a fixed number of runs and zero margin. They explain the gate's limits qualitatively and are not calibrated guarantees for the adaptive protocol of Appendix B.2, which tops five repeats up to seven and reads a regression with a margin of two passes.

C.5 Release chains and aggregate admission

Proposition 3 (Release chains)**.** *(i)* Monotonicity under exact scores. *Let $L_0, L_1, \ldots$ be the libraries accepted by* ScopeEvo *under (A1)–(A2), with nonempty validation sets for every task, exact expected scores at every acceptance check, and the zero-tolerance root rule equation 3. Then $\bar{J}_d(L_{k+1}; V_d) \geq \bar{J}_d(L_k; V_d)$ for every task d and every k, and the mean over the subtree of the edited node increases strictly at every accepted event. Additional acceptance checks, such as the training-set check of §4, only remove acceptances and preserve both statements.*

(ii) Finite samples. *Assume (A3) at every root event, with fixed numbers $n_{d,k}$ of fresh paired runs of d at event k and zero margin. Over K root events, the probability that some candidate lowers the pass probability of some executed descendant on its validation set by at least $\delta > 0$ without being detected is at most $\sum_{k=1}^{K}\sum_{d:\, n_{d,k}\geq 1}\exp(-n_{d,k}\delta^2/2)$.*

Proof. (i) Under exact scores, the gate observes $\bar{\Delta}_d(e)$. A promotion requires $\mathcal{R}(e) = \emptyset$ and $\Delta(e) > 0$, so no $\bar{J}_d$ with $d \in \mathcal{D}(r)$ decreases, their mean increases strictly, and by Lemma 1 every other task is unchanged. A localization leaves the root unchanged; each accepted leaf satisfies $\bar{J}_d(\text{new}) > \bar{J}_d(\text{old}) + \tau_d \geq \bar{J}_d(\text{old})$ and changes no other task, and a rejected leaf changes nothing. A leaf event is the same. Discard, UNDECIDED, and NOHEADROOM change nothing. Induction on k completes the argument. If regressions are read with a tolerance $\rho_d \geq 0$, that is $d \in \mathcal{R}(e)$ iff $\Delta_d(e) < -\rho_d$, a promotion can lower $\bar{J}_d$ by at most ρ_d, and the first inequality holds with $-\rho_d$ on its right-hand side at promotions.

(ii) Let $A_{k,d}$ be the event that candidate k lowers p_d on V_d by at least δ and $d \notin \mathcal{R}(e_k)$. Conditionally on the history and on the candidate, (A3) holds, so Proposition 2 gives $\Pr[A_{k,d} \mid \text{history}, e_k] \leq \exp(-n_{d,k}\delta^2/2)$. Taking expectations and a union bound over k and d gives the claim; the adaptive choice of candidates does not affect it, since each bound holds conditionally. □

Remark 1 (Aggregate-admission counterexample). At the level of exact descendant scores, consider a root candidate that changes two scores from $(0.4, 0.5)$ to $(0.6, 0.4)$, with the first descendant as its proposing set. The mean rises from 0.45 to 0.50, so both LE-AGG and LE-SRC admit it, although the second descendant regresses. The zero-margin rule equation 3 instead returns *localize*.

C.6 FALSE REJECTIONS ON UNCHANGED DESCENDANTS

Proposition 4 (Neutral descendants)**.** *Under (A3) with X_j and X'_j independent and $p'_d = p_d \in (0, 1)$, let $q = p_d(1 - p_d)$. The probability r_{n_d} that the gate returns $d \in \mathcal{R}(e)$ is $\frac{1}{2}\big(1 - \Pr[S = 0]\big)$; it equals $2q - 3q^2$ for $n_d = 2$, about 0.31 at $p_d = 0.5$, and tends to $\frac{1}{2}$ as $n_d \to \infty$. For a set U of unchanged descendants satisfying these assumptions, write r_d for each descendant's flagging probability. With runs independent across descendants, the candidate is refused from the root with probability at least $1 - \prod_{d \in U}(1 - r_d)$.*

Proof. Let $T = \sum_{i=1}^{N} Y_i$ with Y_i i.i.d., symmetric on $\{-1, 0, 1\}$, and $\Pr[Y_i \neq 0] = s > 0$. By symmetry $\Pr[T < 0] = \frac{1}{2}\big(1 - \Pr[T = 0]\big)$. Moreover $\Pr[T = 0] \to 0$ as $N \to \infty$: given the number $M \sim \mathrm{Bin}(N, s)$ of non-zero terms, $T = 2Z - M$ with $Z \sim \mathrm{Bin}(M, \frac{1}{2})$, so $\Pr[T = 0 \mid M]$ is zero for odd M and equals $\binom{M}{M/2}2^{-M} \leq (\pi M/2)^{-1/2}$ for even $M \geq 2$. Hence $\Pr[T = 0] \leq \Pr[M < M_0] + (\pi M_0/2)^{-1/2}$ for every $M_0 \geq 2$. Taking $N \to \infty$ and then $M_0 \to \infty$ proves the limit.

With $p'_d = p_d$ and independent X_j, X'_j, each D_j equals 1 or -1 with probability q and 0 with probability $1-2q$, so the fact applies with $s = 2q > 0$, giving $r_{n_d} = \Pr[S < 0] = \frac{1}{2}(1-\Pr[S = 0])$. For $n_d = 2$, $\Pr[S = 0] = (1 - 2q)^2 + 2q^2$, so $r_2 = 2q - 3q^2$. A candidate is promoted only if no descendant enters $\mathcal{R}(e)$; the descendants in U stay out of it independently with probabilities $1 - r_d$, giving the refusal bound. □

D ADDITIONAL RESULTS

Each figure in this appendix answers one question. Root candidates R1–R16 and leaf candidates 1–9 are defined in the captions of Figures 8 and 4.

D.1 QA CONDITIONS AND SCHEDULES

Figure 6 gives, for every QA condition, the per-domain test-set changes behind the QA panel of Figure 5, the interval against the initial library, and the budget each condition used, together with SCOPEEVO under both of its schedules. Leaf-first, SCOPEEVO starts from Local-only's leaf events and its root branch accepted nothing, so the library it reports in Figure 3 is the one those leaf events produced. Root-first, the root events drew on the budget first, which left the leaf stream five events against Local-only's twelve, and no SearchQA leaf landed; that library's SearchQA score is therefore Root-only's (76.571), while its DocVQA score is above Root-only's (95.095 against 94.084) because the gate localized one DocVQA candidate into leaf candidate 3. What separates the two schedules is thus leaf coverage rather than a placement decision: the root-first library carries the

same promoted root as Root-only plus one localized leaf that raised its domain (§6.3), and its shortfall against Local-only sits almost entirely in SearchQA, where it holds no leaf. On domains that share no cause at the rule level (Appendix D.3), the root search therefore belongs after the leaf events, which is the schedule SCOPEEVO reports.

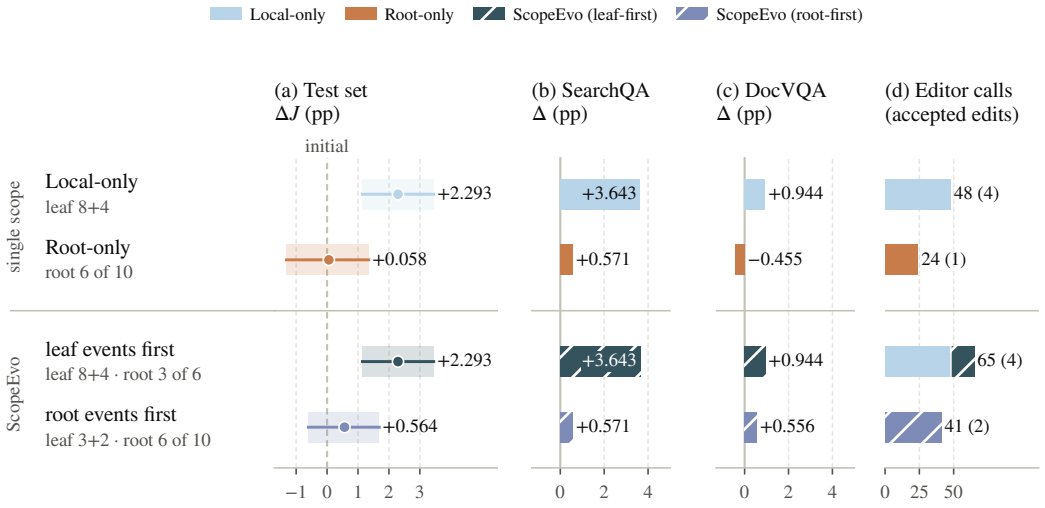

Figure 6: QA conditions on the test set: the change from the initial library in J, with a 95% cluster-paired bootstrap interval, and on each domain, for 1,400 SearchQA items scored by exact match and 374 DocVQA items by ANLS, with J their equal-weight mean; the initial library scores 76.000 on SearchQA, 94.539 on DocVQA, and 85.269 in J. Under each library are its opportunities, leaf events (SearchQA + DocVQA) and root events used, of the cap fixed in advance. The last panel gives editor calls, with accepted edits in parentheses; the leaf-first total includes the 48 calls of the leaf events it starts from, shown in gray.

D.2 SKILLSBENCH PER-TASK RESULTS

Figure 7 gives the per-task entries behind the SkillsBench panels of Figure 5 that come from the full run, in which all descendants were visible to the editors; the unseen tasks are in Figure 4(c). Both halves list the 21 tasks in the same order: the ten consumers of the spreadsheet root, the two consumers of the power-flow root, and the nine tasks whose root no condition changes. The unseen instances were never shown to an editor or to the gate. The training instances were visible to the editors, so the right half measures repair of shown failures, not generalization; it is also where the collateral term of equation 1 becomes visible, on the two consumers that lose passes, discussed in Appendix F, while no consumer of either root regresses on its unseen instances (Figure 7, left). The two power-flow tasks are not scored on training instances under the libraries that carry the full run's power-flow root, whose text contains the optimal values of those two tasks' training instances, so that scoring it there would measure recall rather than repair; the root promoted in the transfer run contains no such value, so grid-dispatch-operator is scored in Figure 4(c).

One shared cause, one root edit. The power-flow family of the full run shows what makes a root edit transfer and what it costs. Both descendants, energy-market-pricing and grid-dispatch-operator, were in the validation set, and both violated the same convention, the branch susceptance model. The promoted root corrected it with a single rule, which lifted the two descendants from $1/8$ to $8/8$ on the validation set and, on unseen instances, from $1/4$ to $4/4$ and from $0/4$ to $4/4$ respectively (Figure 7, left). Local-only reached the same endpoint on the two tasks' validation instances ($0/4 \to 4/4$ and $1/4 \to 4/4$), but it needed three editor calls and wrote two separate leaf rules, one per task; the root route needed two calls and one rule. When descendants share a cause, the root is where that cause is diagnosed once and repaired once, which takes less editing than rediscovering the symptoms task by task and leaves the repair available to descendants that were never diagnosed, as grid-dispatch-operator was in the transfer run (Figure 4(c)).



Figure 7: SkillsBench full run, per task, under the initial library, Root-only, and Local-only: unseen instances (left), the set behind Figure 5(a), and training instances (right), the set behind Figure 5(c). Each run or repeat is one segment, marked as a pass, a pass added or lost relative to the initial library, or a fail. Unseen instances take four runs for the three tasks whose library changed under more than one condition and two for the others; the five reference tasks marked ° have no generated unseen instance and are scored, one run per input, on previously used inputs, and no condition changes their library. Unseen instances were never shown to an editor; training instances were visible to every condition's editor, and the passes lost on them fall on two consumers whose validation runs could not show a regression, one with no validation instance and one at ceiling (Appendix F). ×: not scored under the libraries that carry the full run's power-flow root, whose text contains the optimal values of those two tasks' training instances. J is the equal-weight mean over the 18 tasks of the left half, the 13 with an unseen instance and the five reference tasks, and over the 19 tasks scored under every library, in %; all of the change in the unseen J comes from the two consumers of the power-flow root, and over each library's own scored training tasks J would be 54.0 (21 tasks), 57.9 (19), and 62.7 (21). SCOPEEVO's library adds to Root-only's the court-form-filling leaf, $1/3$ to $3/3$ on the training instances, which gives its gain in Figure 5(c).

D.3 ADMISSION RULES ON THE QA ROOT CANDIDATES

LE-SRC, LE-AGG, and ROOTGATE are functions of the same paired executions, so all three can be computed on every recorded root candidate at no execution cost (Figure 8). The computation asks what each rule does with the candidates this study produced; it does not ask what a different rule would have proposed from the first round on. On the validation set, 11 of the 16 candidates move the two domains in opposite directions, R4–R9 raising SearchQA and R10–R14 raising DocVQA. LE-SRC admits nine of the 16, six of which lower DocVQA on the validation set; LE-AGG admits six, three of which lower one domain. ROOTGATE promotes R1, on which both domains rose on the validation and training sets, returns *localize* on every opposite-sign candidate, and discards R2 and R3, which both baselines admit, because the training set moved their two domains in opposite directions. These counts describe the signal each rule acts on at the validation set's resolution (Appendix B); they do not establish that an admitted candidate would lower the other domain on other inputs.

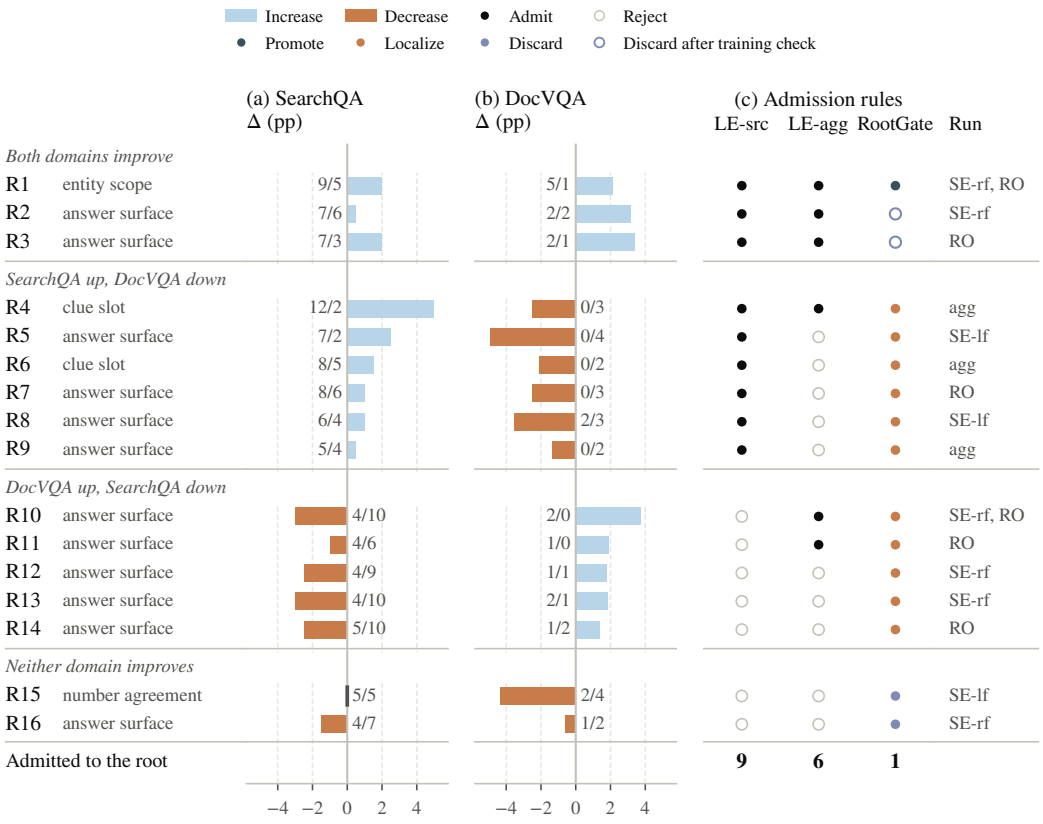

Figure 8: Every QA root candidate scored on the validation set (200 SearchQA and 53 DocVQA items): Δ in pp on each domain, with items improved/regressed beside each bar, and the decision each admission rule reaches on those same executions. Rk is root candidate k, labeled by what the edit governs: entity scope is part versus whole or class versus instance; answer surface is the attested name or span of an otherwise correct answer; clue slot is the role in the clue that the answer fills; number agreement is inflection and singular versus plural. *Run* names the condition that proposed the candidate: Root-only (RO), SCOPEEVO root-first (SE-rf), SCOPEEVO leaf-first (SE-lf), or a run on the same subtree that admitted candidates by the aggregate rule (agg); R1 and R10 were proposed by SE-rf from inputs identical to RO's and are shared. All sixteen were written from SearchQA failures; rows are grouped by the sign of the two changes, and the last row counts the candidates each rule admits to the root. R2 and R3 passed the validation set and were discarded on the training-set check, where the two domains moved in opposite directions.

D.4 LOCALIZATIONS ON THE QA SUBTREE

This section gives the details behind Figure 4(d): the checks on leaf candidate 1 and the causes of the six failed localizations.

The leaf candidate 1 check. Leaf candidate 1 improved SearchQA on the training set ($305 \rightarrow 315$ of 400) and on the 64-item validation subset on which it was validated ($46 \rightarrow 49$, no regressions); the leaf is never delivered to DocVQA, so that domain is unchanged. Of the 19 training items it fixed, 13 have a pronoun or demonstrative subject, the pattern its rule targets.

Failed localizations. Six of the nine were not accepted, for two identifiable reasons. Four DocVQA rewrites could not show a gain on the 53-item validation set, where one item moves the score by up to 1.9 pp: leaf candidates 4 and 5 improved the 107-item training set but not the validation set, and leaf candidates 6 and 7 were near ties on it. The two SearchQA rewrites, leaf candidates 8 and 9, were written into a leaf that already held validated rules; the new text conflicted with the old and lost five items net against a threshold of two. In each case the leaf gate kept the parent, so the library is the one a discard would have produced.

E EXAMPLES OF PLACED EDITS

This section follows two edits from the editor's text to the test result, one through each exit of ROOTGATE that keeps an edit. Example 1 comes from our early exploratory experiments on the QA subtree and serves as phenomenon-level evidence: a root candidate that raised SearchQA but lowered DocVQA on the training set, so it was refused from the root before reaching the validation set, and whose rule, rewritten into the SearchQA leaf, was accepted there (leaf candidate 1 of Figure 4(d)). Example 2 is a promoted root edit that repaired a task the loop never observed (Figure 4(c)). Each box quotes the editor's text verbatim in typewriter type; the bracketed gray lines are ours, and code blocks are omitted.

Example 1 (localize): a refused root candidate, rewritten into the SearchQA leaf

```
[Root candidate, from our early exploratory experiments:  a bullet
proposed for the Question Objective section of the shared root.  It
raised SearchQA but lowered DocVQA on the training set, so the root
check refused it there and it never reached the validation set.]

- Clue-fragment slot test:  if the clue's subject is a pronoun
(He/She/His) or a demonstrative/blank ('this','these', quoted line
ending before 'this'), the slot is that subject's referent or the
word filling the blank.  Name the person/thing the clue describes;
never output an object, title, series name or place mentioned inside
the clue.  Verify:  substitute your candidate into the clue as
subject; if the clue then repeats itself or names a different role,
rebind.

[The editor's own note on DocVQA, submitted with the candidate.]

DocVQA questions name explicit fields ('sales in 2013', 'time of
arrival'), leaving no pronoun subject or trailing blank, so the slot
test never fires and printed-value extraction is unchanged

[Leaf candidate 1:  the rewrite placed in the SearchQA leaf, which
DocVQA never receives.  Accepted by the leaf gate on a 64-item
SearchQA validation subset, from 46 to 49 correct.]

## Clue-Slot Binding

When the clue's grammatical subject is a pronoun (He/She/His) or
demonstrative (this/these), or the clue ends with a blank marker
('this'), the answer is that subject's referent or the blank's
filler--not an object, title, series name, or location mentioned
inside the clue.  Verify:  substitute your candidate into the clue as
the subject or blank-filler; if the clue then reads coherently, the
binding is correct.
```

```
[Two test questions the leaf fixed.  Answer without the leaf -> with
the leaf; the latter is the reference answer.]
His 1944 novel "A Bell For Adano" is set in allied-occupied Sicily
  Sicily -> John Hersey
Her 1991 book said "the undeclared war against American women" is the
"backlash against the women's movement"
  Backlash:  The Undeclared War Against American Women -> Susan
Faludi
```

Example 2 (promote): the section a promoted edit added to the power-flow root

```
[Written from the failures of energy-market-pricing alone;
grid-dispatch-operator, the family's other descendant, entered no
editor input and no validation run.  Code blocks omitted.]
## MATPOWER Equipment Status
When the task is a MATPOWER DC-OPF or market clearing, equipment
status is part of the feasible model.  Do not infer availability from
PMAX, PMIN, reactance, or rating alone.
### Generator status
MATPOWER generator column 7 (zero-indexed) is GEN_STATUS.
- GEN_STATUS > 0:  the generator is online.
- GEN_STATUS <= 0:  the generator is unavailable.
An unavailable generator must not produce energy or reserve, even if
its row has nonzero PMAX or reserve capacity:
[...code block omitted...]
When constructing the generation-cost objective, include production
costs only for online generators.  MATPOWER removes unavailable
generators from dispatch; evaluating an offline generator's
polynomial at zero can incorrectly add its constant coefficient.
[...code block omitted...]
This status rule applies to OPF and market-clearing models with
generator decision variables.  It is not needed for a flow-only
calculation whose bus injections are already fixed.
### Branch status
MATPOWER branch column 10 (zero-indexed) is BR_STATUS. A branch
participates in the network only when BR_STATUS > 0.
[...code block omitted...]
Skip inactive branches when building the network equations,
calculating operational flows, enforcing limits, and reporting loaded
or binding lines.
[...the same edit's changes to the Susceptance Matrix section
omitted...]
```

F DISCUSSION: WHERE THE GATE'S EVIDENCE ENDS

Coverage. Theorem 1(c) says that a descendant with no validation input, or whose validation inputs pass with probability one under both libraries, returns no regression at any number of runs. The promoted spreadsheet root of the SkillsBench full run passed ROOTGATE with no observed regression on its descendants' validation instances, regressed none of its eight consumers that carry an unseen instance (Figure 7, left), and lowered exactly two consumers on their training instances (Figure 7, right): shock-analysis-supply, which has no validation instance because its scorer blocks

input generation, and invoice-fraud-detection, whose runs on its single validation instance all passed under both libraries. Recomputed on the training instances, where the root raises three consumers and lowers two, the per-descendant rule returns *localize* while both baselines still admit it.

QA validation and test. The one promoted QA root, R1, gained about $2\,\mathrm{pp}$ on each validation set; its test-set change had an interval containing zero (Figure 6).

G LIMITATIONS AND FUTURE DIRECTIONS

Four boundaries define what this paper claims. Placement is decided between a root and the leaves of its descendants; intermediate levels of a deeper hierarchy, where an edit could live at a subtree between the two, are the natural next case for the same rule. The gate is built from a few executions per descendant; we do not claim that a gated release chain is free of regressions or that its checks protect inputs outside the validation sets (Appendix F). The topology is given; we instantiate an empty leaf for a task without one and do not merge, prune, or add levels, so learning the hierarchy itself is left open. Scoring is programmatic throughout, by verifiers, exact match, and ANLS, and whether the placement rule behaves the same under an LLM judge is untested.